Design, Evaluation and Application of Software Metrics in Software Testing

A Synopsis submitted to the

Research Degree Committee

for

Doctor of Philosophy

Area of Research
Computer Science and Soft Computing

Submitted by
Anil Bhatnagar

Dr. Lotika Singh
Dept. of Physics & Computer Science

Prof. Gur Saran
Dept. of Mathematics

Head
Department of Physics & Computer Science

Dean
Science Faculty

Research Scholar

Science Faculty
Dayalbagh Educational Institute
(Deemed University)
Dayalbagh, Agra – 282110
(Session: September 2014)

1. Introduction

Testing is integral and indispensable in software development. Testing is necessary to ensure that (a) the software behaves as per stated requirements, including non-functional requirements, and (b) the subsequent revisions of software continue to work without break. Software testing is an expensive part of software development, often accounting for 50% of all development costs [SA 2013]. Test-case creation and maintenance of testcases are labor-intensive tasks. Running of testcases is another time-consuming task, often running for several hours to several days even on server farms.

Modern day software systems commonly involve multiple software components, running on diverse machines and operating systems, including Web technologies and graphical user interfaces. The software implementation uses varied architecture styles such as client-server techniques and repository based. Testing of such complex software systems involves increased emphasis on steps like integration testing, performance testing, usability testing, and interoperability testing as well [OA 2014].

Testcase creation is a very crucial step in the testing process. Testcases must cover the target functionality, including negative cases, must aim to corner cases, and optionally include performance cases. Since testcase creation and maintenance are usually manual processes, the number of testcases grow very quickly and testsuite sizes become very large over the lifetime of software – where many testcases often overlap. However, since testsuite cleanup is a tedious process, the suites are rarely reduced. Thus testcase creation is intellectually demanding and also a highly critical task which has strong impact on effectiveness and efficiency of the testing process [BA 2007]. Automated testcase generation for software systems, therefore, has been deeply researched in recent times and several techniques have been developed [SA 2013].

This research proposal reviews software testing briefly, reviews the various metrics used for software, reviews the existing techniques for automated testcase generation reported in the literature, and reviews search-based software testing in particular. It then proposes to explore using software metrics in multiobjective search based software testing to test software designed with different architectural styles such as client-server software using search based techniques.

2. Software Testing

There are several types of testing needed for software products, such as, Unit testing, Negative testing, Smoke testing, Regression testing, Integration testing, Interoperability testing, Stress testing, Performance testing, Usability testing, etc.[TS 2012]. Each testing type works towards quality assurance of the software and typically a collection of these are executed. Depending on the number of testcases and time taken for running them, and failure tolerance of the development group, some of these tests can also be run continuously, nightly, or even on weekly basis.

A software product or module needs to be tested primarily in two ways [NS 2012]:
(a) Black-box testing – This testing is to check that software provides the functionality per specification. In this testing, only the input-output behavior is checked as per specification without looking into the internal working of the software. Black-box testing does not require the test person to know any implementation details of the software.

(b) White-box testing – This testing is done to check correctness by examining the code. For instance, to check that all paths in the code get covered by testing, and to check exception cases

where the software might fail. White-box testing requires the test person to know the implementation details and internal working of the software.

3. Software Metrics

A "metric" is a measure of some property of a system. In order to be valuable, a metric must be objective, quantifiable, obtainable, and reproducible.

For software development, delivery and maintenance, there are many metrics in use in the industry. These metrics can be categorized as: product metrics, process metrics, and project metrics. Product metrics cover product implementation aspects, namely- size, complexity, design, performance, and quality. Process metrics cover various processes, such as, requirement gathering, specification documentation, defect reporting and tracking processes, etc. Project metrics cover project management aspects such as resource planning, resource assignment, project costs, etc. [MW 2012].

Numerous software product metrics exist in literature and in industry that are used in various software engineering areas implementation, testing, performance tuning, refactoring, etc. Most of these metrics exclude system/vendor libraries and other third-party software. Table 1 below lists commonly used product metrics [LMC 2013].

Table 1: Software Metrics

Software Metric	Measurement	Target
Number of lines of code	Counted as Kilo Lines of code (KLOC)	Cost estimation, Planning
Number of functions		Planning, Task assignment
Comments density	Number of comments per KLOC	Maintainability of software
Number of classes / interfaces		Cost estimation, Planning, Task assignment, Maintenability
Assertions density	Number of assertions per KLOC	Robustness of software
Bugs density	Number of bugs per KLOC	Planning for maintenance cost
Stopper density	Number of stopper bugs per KLOC	Planning for maintenance cost
Code coverage	Percentage of lines/functions covered by running testcases	Testsuite effectiveness
Cohesion	High or low cohesion. Degree to which software in a module works together to provide a functionality	Comprehensibility of software. How related are functions in a class, for example.
Coupling	Loose or tight coupling	Maintainability of software. Tight coupling implies change in one module will need chages in tightly coupled modules as well.
Connascence	Coupling like measure	Connascence can be of many types- function name, type, return values, may be entwined across functions/objects.
Churn	Amount of changes done in a software module over a period of time	Likelyhood of bugs creeping-in is higher. Also, churn of a module will typically impact coupled modules as well.
Cyclomatic complexity	Measures number of linearly independent paths from start to	Complexity, maintainability of software. When cyclomatic complexity is high,

	exit(s) in a source code	recommendations are to modularize the code. Also used in testing.

4. Test Case Generation Approaches

Testcase generation is an intellectually demanding task and it is done mostly manually. However, in the recent decades several techniques have been proposed by researchers for automatic testcase generation [SA 2013].

In general, test cases must be generated by analyzing some information obtained from the software under test itself. While manual testcase generation focuses on feature and coverage aspects, automated approaches require different mechanisms based on more concrete information from the software. Listed below are some of the techniques for software testcase generation.

4.1 Symbolic Execution based test generation

Symbolic execution is a program analysis technique where instead of concrete values, symbols are used as inputs [AS 2009]. Constraints are formed at various decision points in terms of symbolic values and these constraints dictate whether a specific path from the decision point is feasible or not. For example,

```
int assign(int x, int y)
{
      x=y-x;
      if(x > y)
            x=0;
      else
            x=5;
 }
```

Symbolic execution with values x=X, y=Y reveals that constraint x>y is infeasible.

Symbolic execution analysis is also used for creating valid inputs which will ensure that all paths from all decision points can be executed. This is possible because for each feasible path, the path constraints make it possible to decide inputs which will make the path constraints valid. Since symbolic execution analyzes the code, test case generation based on it is belongs to white-box testing.

Symbolic execution has limitations due to path explosion and due to feasible path constraints becoming too complex for realistic programs [SA 2013]. Moreover, in realistic programs several libraries are used where only binary files are available, and hence finding paths in these binary codes poses a significant challenge. Despite these challenges, there exist several proprietary tools and also tools in public domain that generate test cases based on symbolic execution.

4.2 Model based test generation

Model based techniques are formal techniques where a model for software is constructed as an axiomatic model, or a finite state machine.

In the axiomatic model based approach, logic programming is used to find those input values that render the axioms valid [BL 1986].

In the FSM model based approach, software is modelled as an FSM [CT 1978] and then using structural coverage criteria like path coverage, test inputs are selected to create testcases [LD 1996, DK 2006].

Since model based test generation uses model of the code, testcase generation based on it belongs to white-box testing.

4.3 Combinational test data generation

Combinational test data generation technique is a black-box test technique. It does not look into the code, but looks at the various input parameters and their possible values. It then creates combinational sets of the values as testing inputs [CDM 1996, NC 2011].

Most common use of combinational test data generation is combinational interaction technique, where the input parameters from an interactive system (like a GUI or Textual UI) are enumerated along with possible values, and then combination of those are taken as test case data [CMB 2003]. Obvious challenge is that the number of combinations grows rapidly. Constrained combinational interaction technique (CCIT) attempts to contain the combinatorial space by eliminating infeasible combinations [SA 2013]. In CCIT, testcases are added by looking at the input-value possibilities that provide maximum input coverage. Several standard optimization techniques have been used for CCIT test case generation.

4.4 Random testing

Random Testing is a black-box testing approach. It basically creates testcases by using inputs randomly out of the input spaces. Adaptive Random Testing (ART) is one flavour of RT where the new tests are generated as “far” away from successful testcases as possible [CTY 2010, AA 2011]. This is done with the assumption that software has some regions of correct behaviour and some with incorrect behaviour. So, distributing new testcases away from regions is of successful execution more likely to catch defects.

4.5 Search-based software testing

Search based software testing (SBST) approach uses search-based algorithms, usually meta-heuristic based, to select test data or test inputs that maximize a pre-formulated test objective [MP 2004]. SBST approach has appeal because as long as appropriate test objective functions can be created, various search / optimization algorithms can be made to work for testing and for generating test data [MP 2011]. SBST paradigm is quite generic because objectives can include coverage, defects, performance, time, resources, and any other aspects as well [HM 2007].

In SBST, the primary aim is to create a fitness function that captures the test objective. The fitness function is used as a guide for the search based optimization algorithms. The test objectives can be structural (like coverage), functional, non-functional (like performace), etc. This approach is very generic because test objectives can be cast into a fitness function.

Search-based techniques benefit from wealth of searching and optimization techniques, evolutionary algoithms, and heuristic based techniques as well. Hill climbing, simulated

annealing, genetic algorithms, and other evolutionary algorithms have been used for SBST [MP 2004, KG 2009].

If testing targets other objectives along with coverage, then the fitness function will have multiple objectives. SBST approaches handle multiobjective testing/test data generation as well [HM 2007, YT 2011]

5. Identification of Gap

Software metrics for testcase creation

There are many software metrics that can provide better testcase effectiveness and better return on investment (ROI) from each testcase. Very few researchers [NN 2007, AJ 2011] present use of software metrics for testcase generation.

Coverage is just one aspect[GG 2014], many other software metrics can be collectively used. For example, cyclomatic complexity, coupling, and churn can be considered together to define an objective function for use with SBST techniques to get more directed software testing/test data generation targeted towards quicker ROI.

Testing/test generation for different architecture styles

As stated in [SA 2013] here still exists a big gap between real software application systems and practical usability of test case generation techniques proposed by research. There are very few papers like RH 2010, which have attempted to take architecture into account. Software systems have become complex and use various architecture styles such as client-server architecture. There is a need to go for automatic testing/test generation covering different software architecture styles.

6. Proposed Research Objectives

1. To define and evaluate software metrics for search-based software testing/test data generation.

2. To explore the definition and application of software metrics for software testing/test data generation using multiobjective search based techniques and to test software designed with different architectural styles using search based techniques.

7. References

[AA 2011] Arcuri, A. and Briand, L., 2011. Adaptive random testing: An illusion of effectiveness? Proc. Of the 20th International Symposium on Software Testing and Analysis, pp. 265–275.

[AJ 2011] Abdul, J., Subramani, S., 2011. Software Metrics Enhance Test Data Generation and Productivity Measurement. World Congress on Information and Communcation Technologies (WICT), 2011.

[AS 2009] Anand, S., Pasareanu, C. S., Visser, W., 2009. Symbolic execution with abstraction. International Journal on Software Tools for Technology Transfer 11, 53–67.

[AN 2013] Anitha A., A Brief Overview of Software Testing Techniques and Metrics, International Journal of Advanced Research in Computer and Communication Engineering Vol. 2, Issue 12, December 2013

[BA 2007] Bertolino, A., 2007. Software testing research: achievements, challenges, dreams. Proc. Of the 1st Workshop on Future of Software Engineering (FOSE'07), at ICSE 2007.

[BL 1986] Bouge, L., Choquet, N., Fribourg, L., Gaudel, M.C., 1986, Tests set generation from algebraic specifications using logic programming, Journal of Systems and Software 6, 343-360.

[CDM 1996] Cohen, D. M., Dalal, S. R., Parelius, J., Patton, G. C., 1996. The combinatorial design approach to automatic test generation. IEEE Software 13 (5), 83–88.

[CMB 2003] Cohen, M. B., Colbourn, C. J., Gibbons, P. B., Mugridge, W. B., 2003. Constructing test suites for interaction testing. In: Proc. of the 25th International Conference on Software Engineering (ICSE'03), pp. 38–48.

[CT 1978] Chow, T., 1978. Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering 4, 178–187.

[CTY 2010] Chen, T. Y., Kuo, F.-C., Merkel, R. G. and Tse, T. H., 2010. Adaptive random testing: the ART of test case diversity. Journal of Systems and Software 83(1), 60–66.

[DK 2006] Derderian, K., Hierons, R., Harman, M. and Guo, Q., 2006. Automated Unique Input Output sequence generation for conformance testing of FSMs. The computer Journal 49(3), 331–344.

[DM 2013] Debbarma M.K., Debbarma S., Debbarma N., et al., A Review and Analysis of Software Complexity Metrics in Structural Testing, International Journal of Computer and Communication Engineering, Vol. 2, No. 2, March 2013.

[GG 2014] Gregory G., et al., 2014. Moving the Goalposts: Coverage Satisfaction Is Not Enough, 7th Internation Conference on Search-Based Software Testing, Hyderabad.

[GK 2009] Ghani, K., et al., 2009. Comparing algorithms for search-based test data generation of Matlab® Simulink® models, IEEE Congress on Evolutionary Computation, 2009.

[HM 2007] Harman, M., Lakhotia, K. and McMinn, P., 2007. A multi-objective approach to search-based test data generation. Proc. Of the 9'th Annual Conference on Genetic and Evolutionary Computation(GECCO'07), pp. 1098 – 1105.

[KA 2007] Kaur A., Suri B., Software Testing Product Metrics - A Survey, Proceedings of National Conference on Challenges & Opportunities in Information Technology (COIT-2007) March 23, 2007.

[LD 1996] Lee, D., Yannakakis, M., 1996. Principles and methods of testing finite state machines - a survey. In: Proceedings of the IEEE 84(8), 1090–1123. IEEE Computer Society Press.

[LMC 2013] Lee M.C., Chang T., Software Measurement and Software Metrics in Software Quality, International Journal of Software Engineering and Its Applications Vol. 7, No. 4, July, 2013

[MP 2004] McMinn, P., 2004, Search-based Software Test Data Generation: A Survey. Preprint of an article accepted for publication, in Software Testing Verication and Reliability, Wiley 2004

[MP 2011] McMinn, P., 2011. Search-based software testing: Past, present and future. In: Proc. of the 4th International Workshop on Search-Based Software Testing (SBST 2011), pp. 153–163.

[MW 2012] Mandeep Walia, Realizing efficiency and effectiveness of software testing through a comprehensive metrics model, InfoSys Whitepaper, 2012.

[NC 2011] Nie, C. , Leung, H. , 2011. A survey of combinatorial testing. ACM Computing Surveys 43 (2), 1–29.

[NN 2007] Nagappan, N., Ball, T. (2007) Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. IEEE First International Symposium on Empirical Software Engineering Measurement.

[NS 2012] Nidhra S., Dondeti, J., 2012, Black-box and White-box testing techniques- A Literature Review, International Journal of Embedded Systems and Applications, Vol.2, No.2, June 2012.

[OA 2014] Orso, A., Rothermel, G., 2014. Software Testing: A Research Travelogue(2000-2014). Workshop on Future of Software Engineering (FOSE'14).

[RH 2010] Reza H., Lande S, 2010. Model Based Testing Using Software Architecture, Seventh International Conference on Information Technology: New Generations (ITNG), 2010

[SA 2013] Saswat Anand et al., 2013, An Orchestrated Survey on Automated Software Test Case Generation, Antonia Bertolino, J. Jenny Li and Hong Zhou (Editor/Orchestrators), Pre-print submitted to Journal of Systems and Software.

[TS 2012] Trivedi S.H., 2012, Software Testing Techniques, International Journal of Advanced Research in Computer Science and Software Engineering.

[YT 2011] Yano, T., Luis de Sousa, F., 2011. A Multi-Objective Evolutionary Algorithm to Obtain Test Cases With Variable Lengths, Proc. Of the Annual Conference on Genetic and Evolutionary Computation(GECCO’11).